# FPGA ACCELERATION OF CNN TRAINING

A Thesis
Presented to
The Academic Faculty

by

Kruttidipta Samal

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2015

# FPGA ACCELERATION OF CNN TRAINING

Approved by:

Dr. Marilyn Wolf, Advisor
School of Electrical and Computer Engineering
*Georgia Institute of Technology*

Dr. Tom Conte
School of Electrical and Computer Engineering
*Georgia Institute of Technology*

Dr. Saibal Mukhopadhyay
School of Electrical and Computer Engineering
*Georgia Institute of Technology*

Date Approved: 12.04.2015

# ACKNOWLEDGEMENTS

I wish to thank my advisor Prof. Marilyn Wolf for her guidance and support without which this thesis couldn't have been complete. I am very grateful to my professors and teachers both here at Georgia Tech and in India who have imparted me with the knowledge and confidence to complete this work. I would also like to thank my friends and colleagues at Georgia Tech for discussing and critiquing my ideas.

Finally I am most grateful to my parents, elder brother, family and friends for their moral and financial support that inspired me to complete this work.

# TABLE OF CONTENTS


| | Page |
|---|---|
| ACKNOWLEDGEMENTS | iii |
| LIST OF TABLES | v |
| LIST OF FIGURES | vi |
| LIST OF CODE SNIPPETS | vii |
| SUMMARY | viii |
| CHAPTER | |
| 1 INTRODUCTION | 1 |
| 2 VISION IS COMPLEX | 4 |
| Learning Algorithms | 6 |
| Functionality | 11 |
| 3 CONVOLUTIONAL NEURAL NETWORKS (CNN) | 19 |
| 4 DRAM SIMULATION | 29 |
| Architecture | 31 |
| Result | 33 |
| 5 FPGA DESIGN OF CNN | 40 |
| Architecture | 41 |
| Memory Storage Calculations | 46 |
| Utilization | 47 |
| Loop Iteration | 48 |
| 6 CONCLUSION | 50 |
| APPENDIX A: Code Walkthrough | 52 |
| REFERENCES | 56 |

# LIST OF TABLES


Page

Table 1: Layer-wise algorithm distribution 22

Table 2: Utilization Table in [34] 48

Table 3: Utilization Table in Virtex 7 Ultrascale xcvu440 48

Table 4: Loop Latency with single Processing Element 49

Table 5: Loop Latency with 2916 Processing Element 49

# LIST OF FIGURES


Page

Figure 1: Layered Architecture of Neo-cortex 4

Figure 2: Artificial Neuron 8

Figure 3: Stacked Autoencoder 8

Figure 4: HMAX Model 15

Figure 5: Transforming Autoencoder 15

Figure 6: LCN in space 16

Figure 7: Left: Biological neuron Response Right: Common Artificial neuron response 17

Figure 8: Yann LeCunn's CNN (first successful artificial CNN) 19

Figure 9: AlexNet Architecture 21

Figure 10: Zeiler's CNN architecture 23

Figure 11: Coupled Convolution and Deconvolution Network 23

Figure 12: RCNN 24

Figure 13: Google LeNet 28

Figure 14: Flowchart of DRAM Simulator 31

Figure 15: Sample Simulator output of single channel DRAM running on first convolution layer trace 36

Figure 16: Sample Simulator output of double channel DRAM running on first convolution layer trace 37

Figure 17: Sample simulator output of second convolution layer trace 38

Figure 18: Conversion and storage of input image in BRAM blocks 41

Figure 19: Architecture designed in [33] 42

Figure 20: Designed FPGA Architecture 43

# LIST OF CODE SNIPPETS

Page

Code Snippet 1: HLS code for FPGA with single Executing Unit 52

Code Snippet 2: Sample code for BRAM port and block declaration 53

Code Snippet 3: Sample Code containing data transfer and computation logic to generate 2916 parallel processing units 54

# SUMMARY

This thesis presents the results of an architectural study on the design of FPGA-based architectures for convolutional neural networks (CNNs).

We have analyzed the memory access patterns of a Convolutional Neural Network (one of the biggest networks in the family of deep learning algorithms) by creating a trace of a well-known CNN architecture and by developing a trace-driven DRAM simulator. The simulator uses the traces to analyze the effect that different storage patterns and dissonance in speed between memory and processing element, can have on the CNN system. This insight is then used create an initial design for a layer architecture for the CNN using an FPGA platform. The FPGA is designed to have multiple parallel-executing units. We design a data layout for the on-chip memory of an FPGA such that we can increase parallelism in the design. As the number of these parallel units (and hence parallelism) depends on the memory layout of input and output, particularly if parallel read and write accesses can be scheduled or not. The on-chip memory layout minimizes access contention during the operation of parallel units. The result is an SoC (System on Chip) that acts as an accelerator and can have more number of parallel units than previous work. The improvement in design was also observed by comparing post synthesis loop latency tables between our design and one with a single unit design. This initial design can help in designing FPGAs targeted for deep learning algorithms that can compete with GPUs in terms of performance.

# CHAPTER 1

# INTRODUCTION

Recent developments in deep learning has led to its application in tasks that were earlier possible only with traditional handcrafted methods such as SIFT (Scale Invariant Feature Transform) [1], HOG (Histogram of Gradients), BoW (Bag of Words) [2] and LHoP (Learning Hierarchy of Parts) [3]. Deep Learning methods are vestiges of developmental stages of cognition. As biological vision systems are very robust even in depraved conditions, learning features in line with biology, can lead to better performance of artificial systems especially in divergent conditions which are pain points in modern vision systems. These divergent conditions are (but not limited to) object recognition in partial occlusion, extraction of form from motion, inference of surficial properties from reflectance etc. In current review we are concerned with how to relate two images. In particular, we are concerned with tracking a moving object and/or infer the transformation of an object in an image sequence. Current state of the art in Computer Vision for tracking is Bayesian method called 'Particle Filter'. But we will approach this topic by studying the visual pathways in visual cortex and juxtapose them with the state of the art in deep learning. Extraction of motion and extraction of transformation have traditionally been entirely separate from one another, this will be evident from some of the networks that will be presented, but the same isn't true in biology, as it is not very functionally structured. Of course, the best method would be to figure out affine transform of an image, this can take care of all kinds of transformation, but as the artificial system receives an intensity value per location, its very difficult to associate an

intensity to an object and then calculate the transformation factors. Despite all these challenges, a recent review of three DARPA funded neuromorphic vision systems [4] confirmed that biologically inspired systems are much more energy efficient and have almost nearly and sometimes better performance.

All the three projects model the human vision systems from a systems perspective. But as the systems themselves aren't modeled according to biology, they loose on efficiency. Maintaining scale invariance is an issue, this is mitigated by these systems by maintaining multiple scaled versions of the same object feature. Also these features are hand-crafted and not learned. Although the model from Penn State and USC have efficiently inculcated the concept of 'Attention' into their model, this is based on saliency maps, which don't take care of 'objectivity' of a region, thus their attention region may not have a rigid structure or form. Finally their attended region is compared to already stored BoW or LHoP to detect an object and use a particle filter to track. Thus, even though these neuromorphic systems are good, they can be much better, if modeled from studies in neuroscience and deep learning.

Therefore, this report attempts to discern those biological systems that are influential in tracking and/or object detection/recognition while comparing them with the traditional and state of the art technologies in Computer Vision, both at an algorithmic level as well as at system level. It starts with basic cortical neuronal connections vis-à-vis sparse coding and ICA. Then it moves into neural pathways and systems that are associated with it, following it with a brief study on deep learning architectures and documenting their evolution in tandem with biology. One of the popular system implementations is analyzed with respect to its memory accesses and an FPGA design is

presented whose performance is comparable to prevalent implementations. The report ends by delineating those aspects of FPGA that should be enhanced to get better performance.

# CHAPTER 2

# VISION IS COMPLEX

This section will deal with the biological structures and phenomenon underlying human vision. Observing structure in- Connection and Functionality separately would present more coherent understanding of the same. We would later observe both in conjecture, when they are implemented in deep learning systems.

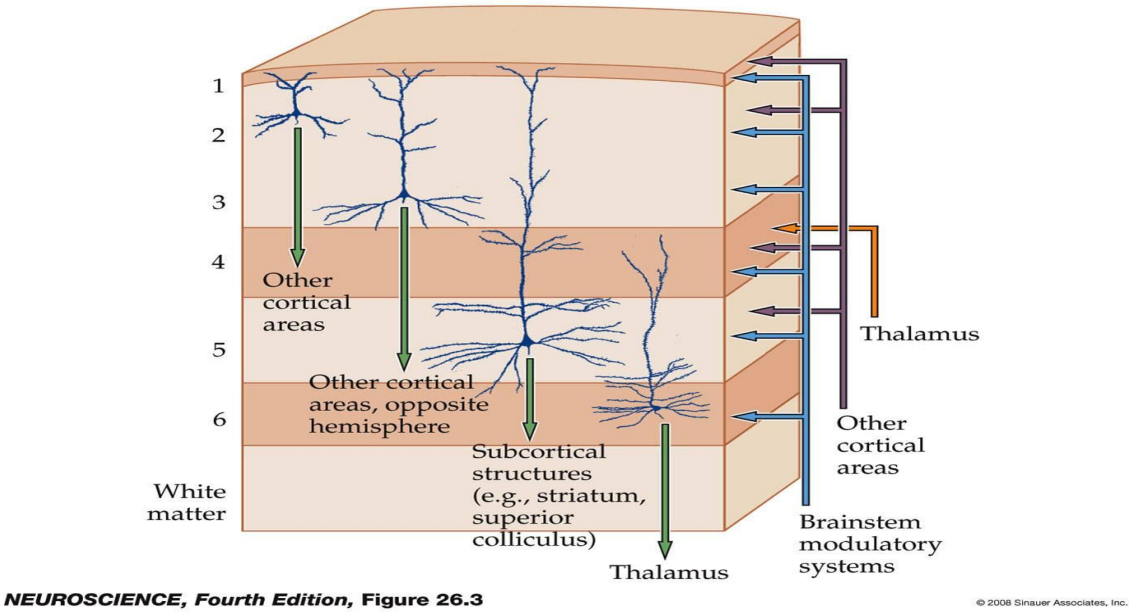



Figure 1: Layered Architecture of Neo-cortex

Neocortex Layered Architecture- Being the youngest in evolutionary time-cycle, and biggest in volume share of the brain, neo-cortex is considered to be the center of all higher cortical functions. One of the additional evidences suggesting its efficacy is the fact that humans have almost twice as many neurons in their neocortex than chimpanzees, our predecessor in cognitive development. One of the prevalent hypotheses for explaining

cortical computation- single algorithm theory [5] suggests the presence of a single algorithm applied in each layer of the cortex. Considering cortex is made up stacked layers with uniform structure, it is safe to assume that this architecture is a layered, hierarchical and uses repetitive application of the same algorithm. This general structure of neocortex is independent of function i.e. visual cortex is almost similar in structure to auditory cortex. It is well documented that there are roughly 6/7 layers in neocortex. The input from thalamus enters into Layer 4, which is fed into layers 2/3 to 1, and there are feedback connections from these layers to layers 5,6. The number of feedback connections in cortex is almost seven times that of feedforward. This kind of architecture is very effective for reducing redundancy in the incoming signal. One of the models that explore this idea is predictive coding model [6] that uses a sparse autoencoder like algorithm to predict the value of a pixel from the intensity values of surrounding pixels, when applied to natural images. But similarly there is another model called HTM (Hierarchical temporal memory) [7]–[9] by Jeff Hawkins, inspired from Lee and Mumford's Bayesiam model of cognition, which does a similar kind of prediction in both space and time. While the aim of Rao and Ballards's predictive coding model was increasing coding efficiency by using feedback, the aim of HTM is to improve prediction. But one of the issues with HTM is that it uses a SDR- Sparse Distribution Representation, to convert raw signals into encoded ones where each bit has a semantic meaning, before they can be fed into the learning network. As our aim is to study visual processing in cortex, we cannot venture further into HTM, owing to its above limitation. From the above connections, we can infer what kinds of learning algorithms may be employed in our cortex.

## Learning Algorithms

As discussed, most of the popular learning algorithms are responsible for reducing dimension of representation and result in successively sparse representation of a given data. Independent Component Analysis is one such algorithm, it represents the data in term of weights of independent components which are linearly independent of each other. Thus the dimension of internal representation has to be ideally equal to that of the data. But as the basis vectors/functions are linearly independent, progressive application of ICA on the data will weed out redundancies and thus induce sparseness. But one of the conditions for calculating ICA is $W*W^T = I$, where W= weight matrix of the basis vectors and I is the identity matrix. This problem is intractable. One of the work-around is called RICA- Reconstruction ICA [10] by Andrew Ng's team at Stanford which adds a reconstruction term while calculating the cost and iteratively tries to minimize this cost function.

ICA optimization: $\min_W \sum_{i=1}^{m} \sum_{j=1}^{k} g(W_j x^{(i)}),\ subject\ to\ WW^T = I$

where, g(.) is a non-linear convex function like log(cosh(.)), W=weight matrix $W \in \mathbb{R}^{k \times n}$, k is the number of features and $W_j$ is one feature in W, 'm' is total number of data points available.

ICA Reconstruction Optimization:

$$\min_W \frac{\lambda}{m} \sum_{i=1}^{m} \left\| W^T W x^i - x^i \right\|_2^2 + \sum_{i=1}^{m} \sum_{j=1}^{k} g(W_j x^{(i)})$$

Another such algorithm is Sparse Autoencoder. Unlike ICA, SA's internal representation's dimension is more than that of data. Thus the basis vectors can no longer be linearly independent, but the sparsity constraint enforces the basis vectors to be mutually orthogonal. Similar to RICA, we reconstruct the input from output by applying the same weights to the output and calculate how off our reconstructed input is from original input. According to reconstruction errors, the weights of the layer are changed. This can be done by backpropagation [11] using gradient descent where we calculate the effect (or gradient) of output based on each weight, which controls the change for that weight, this is repeated every iteration.

Both of the above algorithms are linear representations of given data, but its conceivable that all data can be represented as non-linear combination of basis vectors [11], [12]. In order to introduce nonlinearity and simultaneously maintain the ease of operation of the above algorithm, they are implemented as stacked layers each running the same algorithm on the output of the lower layer. Thus making the entire set-up non-linear. One of the demerits of this set-up is that now gradient descent algorithm can be stuck in localized minima, while this problem will always persist, we can use SGD-Stochastic Gradient Descent and dropout [12] etc. to avoid it.

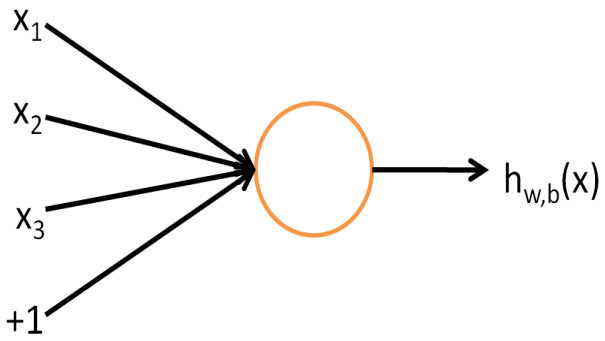



Figure 2: Artificial Neuron

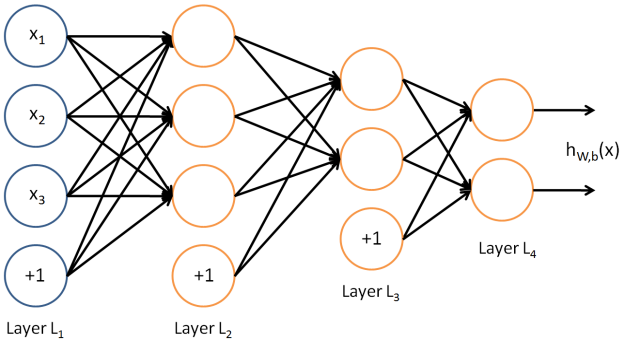



Figure 3: Stacked Autoencoder

Each neuron in a SA network has an activation function such as- $h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^{n} W_i x_i + b)$. Where, $W_i$ are the weights of a network, $x_i$ is the input and b is a bias.

Following equations are frequently used in SA

Cost Function to be optimized - $$J(W, b; x, y) = \frac{1}{2} \|h_{W,b}(x) - y\|^2$$

$$J(W, b) = \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)})\right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)}\right)^2$$

Here, $\lambda$ is a regularization factor and the second term of the above equation (also called regularization term) is used to avoid a condition where weights are tuned to predict

accurately for training data only and don't perform well for unseen data, a condition called overfitting.

Update Rule for weights and biases, based on value of cost/error function, performed on every iteration on a data element –

$$W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$
$$b_i^{(l)} := b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b)$$

Gradient of cost function determining the of change of network parameters –

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \left[ \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}) \right] + \lambda W_{ij}^{(l)},$$
$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(i)},y^{(i)}).$$

Sparsity constraint –

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[ a_j^{(2)}(x^{(i)}) \right]$$

$a_j^2$ is the activation of a particular node j for an input $x^{(i)}$

This term is equated to a sparsity parameter $\rho$, which is usually kept at very low value. This is such to make sure that the total number of activations in the network is kept at a minimum. Now, the divergence between these two terms has to be kept at a minimum. This divergence is captured by Kullback-Leibler (KL) divergence, which is of the form –

$$\sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$$

This term is added to our earlier cost function to result in this final cost function –

$$J_{\text{sparse}}(W,b) = J(W,b) + \beta \sum_{j=1}^{s_2} \text{KL}(\rho || \hat{\rho}_j)$$

*Further implementation details beyond scope of this report, please refer to given citation for further details.

As the architecture discussed above is layered, it is very difficult to optimize the network by doing gradient descent; calculating gradient for each of the weights in the network and backpropagating the error caused at the top-most layer into each node in every lower layer. One of the hacks commonly used is called 'greedy learning', i.e. learning each of the layers one at a time in an unsupervised manner by calculating the reconstruction error and adjusting the weights in that layer based on this error only. This is called pre-training. As the size of such networks, like the one used for generic object detection, can be very large, there is a risk of overfitting, where the network performs well for training data but not for general data as the parameters are optimized only for data seen during training. One of the techniques used to avoid such scenario is called drop-out, developed by G.Hinton's lab at Univ. of Toronto. According to drop-out [11], [12], contribution of any node in the network is dropped with a probability of 50%. This introduces a level of stochasticity to the network and forces the network to learn more generic form of representation.

Both ICA and Autoencoder are deterministic methods of learning. Similarly there are other families of learning algorithms that are probabilistic. Most popular of these are called RBM-Restricted Boltzmann Machine[13]. In RBMs there are 2 visible sub-layer-input and output and 1 hidden sub-layer. State of each hidden sub-layer node is calculated by a linear weighted sum of the input, this is synonymous to calculating the probability of activation of a hidden node conditioned on a given input. The same is done for output sub-layer. RBM is an energy model, i.e. the nett. Energy of the network is calculated based on a particular state of all the nodes in the layer. Optimization can be achieved by changing the weights such that the energy of the network is minimized. But this requires calculating aposteriori of the probability of an input node activation conditioned on state of the hidden layer. But as this data is not available, it is approximated using iterative gibbs sampling, this takes a long time and accuracy depends on the number of iterations. Furthermore, gradients or change in weights to minize the energy function is done using Contrastive Divergence, which is similar to gradient descent in Autoencoders.

For the rest of the report we will concentrate on autoencoders for learning.

## Functionality

One of the theories explaining cortical computation, other than the single algorithm theory, is functionally Modular design [5]. Such a design states that the brain has a very modular design and consists of specialized regions not just in high level cortexes (such as visual, auditory etc.) but also within a cortex. One of the evidences supporting such a claim would be the fact that- V1 is responsible for filtering edges (although about 30% of cells are thought to be direction selective), IT region has cells which are selective of faces (and also some non-face objects), similarly hippocampus has cells which are called ‘place

cells' which are responsive to certain location in space. These are suggestive of a task-wise breakdown of processing. Thus, an agreement between the single algorithm theory and modular design would be an architecture that would apply the same algorithm on different data (either pre-processed with the same algorithm, raw data or data from different processing area) and result in functionally different results.

One of the traits that is required for above design is hierarchical and multi-modal structure. Most of the work in deep learning has been concentrated on hierarchical structures. Epitome of multimodality can be observed by analyzing Basal Ganglia part of the brain, which gets input from visual and auditory cortexes, limbic system, and amygdala that is responsible for emotions. As this system is responsible for learning, this type of multi-modal operation facilitates in coherence and also support the long standing belief that emotions can facilitate learning and recall.

Observing the esoteric nature of generic multi-modality in the brain, its better if we concentrate on hierarchy for now. The most prevalent explanation of visual system is the two pathway theory [14] - The output from retina cells (bipolar and ganglion cells) go to LGN (Lateral Geniculate Nucleus, also responsible for consolidation of signals coming from both eyes), forms a nerve bundle and fed into V1. Operations such as local contrast normalization and whitening to reduce redundancy and shredding of unnecessary information are done in LGN and retina cells. V1 cells are traditionally thought of as edge detectors or Gabor wavelet detectors. These cells are distributed in space, such that a neighborhood of cells is responsible for a patch in space. They are localized in orientation and frequency space, i.e. cells responsible for same patch in image are

responsive to different Gabor orientation and spatial frequency. The output of V1 is fed into two different pathways- Ventral and Dorsal.

Ventral Pathway (V1-V2-V4-IT) - This is called the 'what' pathway. It is responsible for object recognition and thus has to deal with invariance such as- spatial, rotational and partial occlusion. As V2 and V4 are not easily accessible, not a lot has been documented on the inner workings of these layers. But this entire pathway can be thought of as successively capturing complex structures of objects such as edges, shapes, textures etc. As, there has been a lot of psychophysical studies in this pathway especially in V1 and IT, the amount of literature available is huge and the reason why deep learning community is the most attracted to this pathway. There are 3 models which have been inspired from this pathway- HMAX pooling, LCN and reLU.

HMAX – Developed by Riesenhuber and Poggio [15]. This is a simple method to tackle spatial invariance. Same object occurring at different locations in the scene don't have to be learnt explicitly, implying a learning technique, independent of location. Therefore, HMAX- hierarchical max pooling, is a layered architecture, where each layer's node pools over a predetermined region of the lower layer. This pooling can be a mean, max or any other operation. Biologically, this can be similar to keeping the most active response within a population, symbolizing a kind of down-sampling. This not only gradually decreases the total number of feature maps/filter maps/information to be stored and processed in the network but as the most active set of nodes in one corner of the image can come to the center after repetitive application of this pooling, provided that they are

active enough, and thus final learning layer can learn this pattern of activation, without regards to where in the scene it occurred.

While HMAX has been very successful in translation invariance there has been a lot of criticism as it leads to loss of a lot of details. Traits belonging to the same object can be lost due to this winner takes all configuration. Incase of multiple objects, due to max pooling, the details of one of the objects is completely lost or even worse details of both objects can be skewered to form a high level activation that is characteristic of neither leading to faulty classification. Due to lack of preservation of any low level detail, it becomes very hard for a high level layer to detect multiple objects. There has been some work in this regard such as capsule theory [16], where the detail propagated to higher layer includes the absolute position of the pattern activating that. But one of the problems with this idea is that, the algorithm can get confused if the same pattern occurs several times in the same scene, thus confusing the absolute location of the pattern. Also, this design recommends each node to have access to entire image, thus removing the concept of local RF, which is not inline with biology and is computationally expensive.

While dealing with spatial invariance with local RF, it is intuitive to assert that the patterns occurring in one part of the image can also occur at different part of the image. Thus the patterns learnt from one part can be reused all over the image. This led to tied weights where the filter parameters learnt from one part of the image is applied all over to get feature maps, a different but intermediate approach used in [17], with filters that lie within a small range have tied weights. But by sharing filter parameters, we need more number of filters to incorporate all variations possible.

One of the alternatives to HMAX, is maintaining multiple copies of feature maps/response activation at multiple scales and using the one which gives the best detection. This was heavily used for traditional computer vision techniques such as SIFT/HOG and also in some earlier implementations of HMAX, but as the number of layers in NN grew bigger, it had to be dropped as storage became an issue.

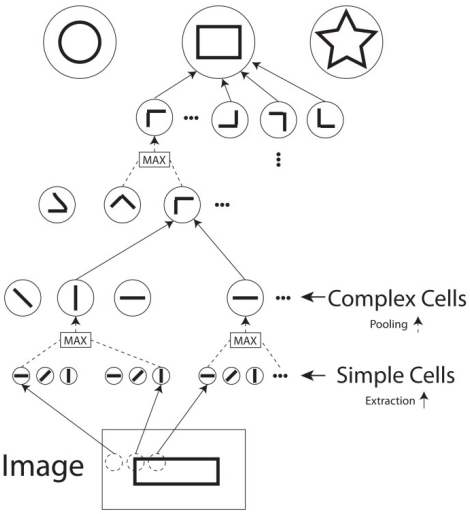



Figure 4: HMAX Model

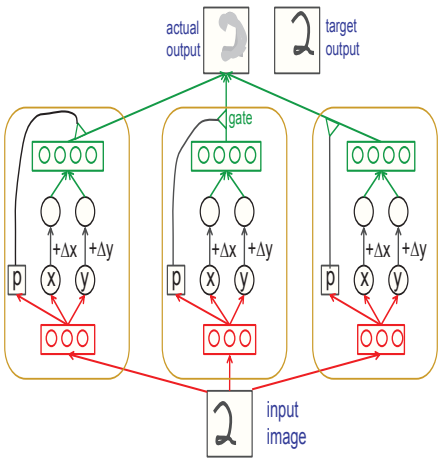
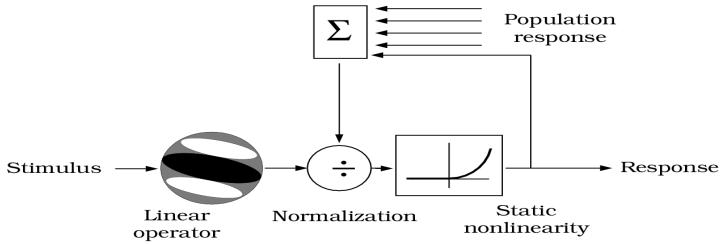



Figure 5: Transforming Autoencoder

LCN – Local Contrast Normalization [18]. This is also called divisive normalization, where the value at a feature response map is divided by the mean response of all other values of that feature map in a particular neighborhood. This decreases redundancy in response maps. This normalization can also be subtractive, where a pixel in a feature map is subtractive by the Gaussian weighted mean of responses of its neighbors. In either case, this technique is similar to predictive coding where, reducing spatial redundancy increases coding efficiency, because features usually extend over long distances beyond local RFs. But there can be redundancy reduction is across filters, as responses in neighborhood tend to capture similar features (neighboring filter's parameters are closely associated, as in MT columns). This normalization can also be applied across channels, e.g. Red, Green and Blue channels. Most of the models of LCN in use rigid and do not learn normalization from natural image statistics that can be an area of future investigation.

$$f(u_f^{x,y}) = \frac{u_f^{x,y}}{(1 + \frac{\alpha}{N^2} \sum_{x'=\max(0,x-\lfloor N/2 \rfloor)}^{\min(S,x-\lfloor N/2 \rfloor+N))} \sum_{y'=\max(0,y-\lfloor N/2 \rfloor)}^{\min(S,y-\lfloor N/2 \rfloor+N))} (u_f^{x',y'})^2)^{\beta}}$$




$u_f^{x,y}$
$= value\ of\ response\ map\ f\ at\ co-ordinate\ (x,y)$

N = Meighborhood to be normalized.

f = output of divisive normalization.

Feature Map normalizing in one feature map in single channel.

Figure 6: LCN in space

reLU – Rectifying Linear Unit [12], [19], [20]. Untill now we have assumed the output of a model neuron is the weighted sum of its lower node's activation, but in biology, the neuron fires only when the sum of all its weighted inputs is above a particular threshold. This threshold often is a soft threshold and this response function isn't a step function, rather is quite smooth, such as a 'tanh' or logistic functions these are called activation functions. When used in a network, this function can create a problem during back-propagation. In gradient descent, the updated weight is calculated by subtracting the current weight with a value proportional to weighted error from the layer above. Now, the gradient of a tanh/logistic function is zero valued except in the neighborhood of zero. This implies that, if the error values are large (which usually occurs during initial phase of the training), the correction term is going to be zero, due to zero value of activation gradient. Thus, the network may never optimize, or might take very long to do so, this is called as decaying gradients problem. This is mitigated by changing the activation function to a function whose gradient is non-zero even at large positive values such as max function (reLU(x) = max(0, x)) [20]. This function shows the kind of asymmetry that is evident in cortical neurons as shown in the figure 7.

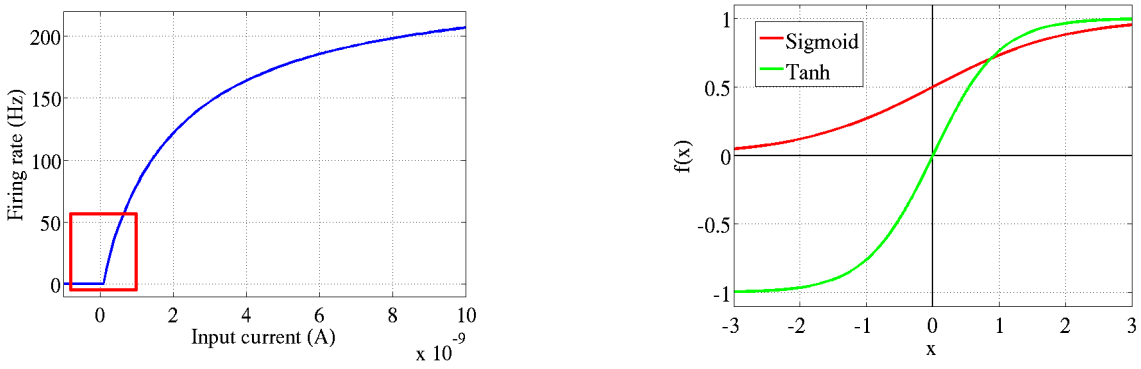



Figure 7: Left: Biological neuron Response Right: Common Artificial neuron response

Now we can put all of the above techniques together to form a network that can learn patterns. Of all the neural networks (NN), one that has recently become most successful is called Convolutional Neural Network.

# CHAPTER 3

# CONVOLUTIONAL NEURAL NETWORK (CNN)

CNN (Convolutional Neural Networks) [21] are the state of the art in computer vision. From signal processing we know that Convolution can be interpreted as inverse correlation. Thus, convoluting an image patch with a filter is similar to finding how similar that patch is to the feature selective to that filter. However the neural basis of convolution isn't well document, but by observing psychophysical experiments we can infer of a convolution-like processing underlying at least in early stages of ventral pathway. Artificial CNNs were successfully applied to handwritten digit recognition in [22], their first implementation to real world problem. But recent developments that facilitated building of large networks, lead to a rejuvenation of the same for natural image processing. The concept of a CNN is pretty generic- Stacked layers of convolution followed by stacked layers of unsupervised learning, which can then be fed into a classifier.

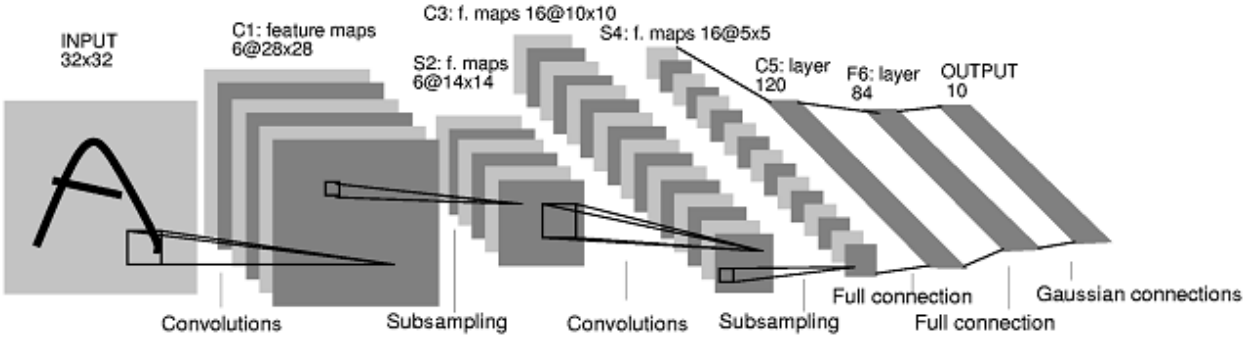



Figure 8: Yann LeCunn's CNN (first successful artificial CNN)

LeNet – As can be observed from figure 8 above the feature maps become smaller in successive stages in the network and so does the receptive field of each node/neuron. This is due to the intermittent subsampling/pooling layers, these are mean pooling rather than max pooling. But even though the RF of neurons in higher layers is small, it actually represents large regions in image space, this can be observed by de-sampling all the pooling layers. The big receptive fields of higher neurons help to achieve some spatial invariance. Here, the filters can be thought of localized autoencoders, and error propagation is also similar. Also, the convolutions are not localized in 2D but are 3D, this results in uneven spread of feature maps across different layers, which is not very intuitive. However, this network is not nearly as big as the state of the art. This is because of all the reasons discussed previously, such as gradient/error decay during backpropageation, risk of overfitting, overdependence on initial conditions (due to absence of reLU) and the most important lack of computational power/ huge latency due to lack of parallelism in implementation.

AlexNet – Developed by Alex Krizeskvy at University of Toronto [23], won the ImageNet object recognition challenge in 2012. The architecture, shown in figure 9, can be considered a scaled up version of Lenet with all the modifications such as max pooling, reLU and dropout with pre-training. The implementation was run on a GPU that decreased execution time to few days, depending on how much pre-training is done.

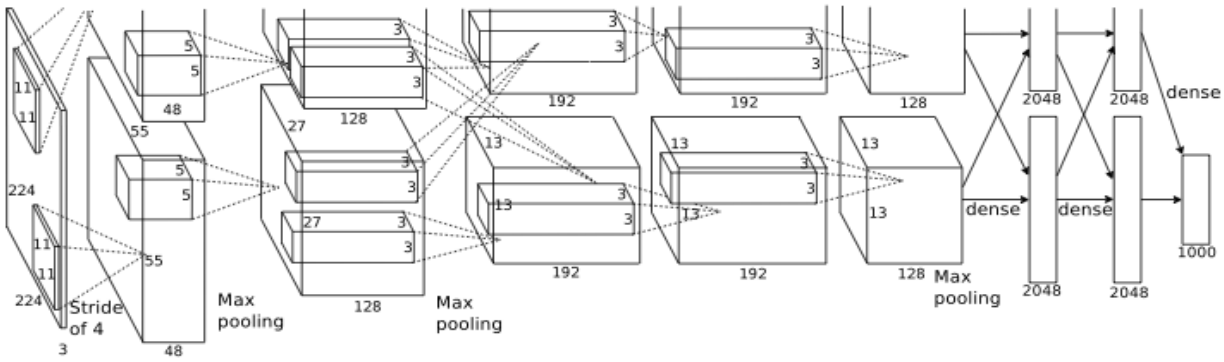



Figure 9: AlexNet Architecture 2012

This network had about 60 million parameters, 650 thousand neurons. With these many parameters, overfitting was avoided using dropout. It was trained on 1 million images from Imagenet database. Each of these images was 256x256, but each image was divided into 10 images of 224x224 size. Thus the network was trained on 10 million images. This model has five convolution layers, each followed by a max pooling and reLU layer. The final output is connected to two consecutive fully connected layers. Regularization was used to avoid overfitting in these fully connected layers. Regularization implies adding another term to cost/error function that adds an extra penalty on weights, thus limiting them from achieving very high values. This is similar to imposing a sparsity constraint to the weights. The pooling regions were overlapped, thus capturing local translation invariance. The final output layer had 1000 nodes, and hence the layer can ideally discriminate between 1000 categories (equal to total number of classes in Imagenet 2012). The target of the final layer was to maximize log probability of correct prediction across all image classes. Images were trained in mini-batches rather than one-by-one and weights were updated after each mini-batch was processed.

Table 1: Layer-wise algorithm distribution

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| | Cov1 | Cov2 | Cov3 | Cov4 | Cov5 | fc6 | fc7 | fc8 |
| | Data<br>↓conv<br>conv1 | norm1<br>↓conv<br>conv2 | norm2<br>↓conv<br>conv3 | conv3<br>↓conv<br>conv4 | cov4<br>↓conv<br>conv5 | pool5<br>↓IP<br>fc6 | fc6<br>↓IP<br>Fc7 | fc7<br>↓IP<br>Fc8 |
| Data | conv1<br>↓relu<br>conv1 | conv2<br>↓relu<br>conv2 | conv3<br>↓relu<br>conv3 | conv4<br>↓relu<br>conv4 | Conv5<br>↓relu<br>conv5 | fc6<br>↓relu<br>fc6 | Fc7<br>↓relu<br>fc7 | Fc8<br>↓softmax<br>prob |
| | conv1<br>↓pool<br>pool1 | conv2<br>↓pool<br>pool2 | | | conv5<br>↓pool<br>pool5 | fc6<br>↓dropout<br>fc6 | Fc7<br>↓dropout<br>fc7 | prob<br>↓dropout<br>label |
| | pool1<br>↓norm<br>norm1 | pool2<br>↓norm<br>norm2 | | | | | | |

IP: Inner-product

This is the state of the art in Computer Vision for Object recognition, achieving a top-5 error rate of 17% and top-1 error rate at 37.5%. Subsequent models by others have improved upon this model mainly by changing 2 parameters-

1. Changing the size of filter kernels and stride of pooling layers.
2. Changing hyperparameters such as increasing number of parameters- filter and fully connected neurons.

Zeiler's CNN – This [24] was presented in CVPR 2014 and is similar to AlexNet. The only change was that the size of filter in 1st layer was decreased from 11x11 to 7x7, and stride of filter was decreased from 4x4 to 2x2. The convolution in layers 3,4 and 5 ware fully connected unlike AlexNet, as the authors didn't partition their data for GPUs. These changes improved the performance of their network from AlexNet by 1.4%.

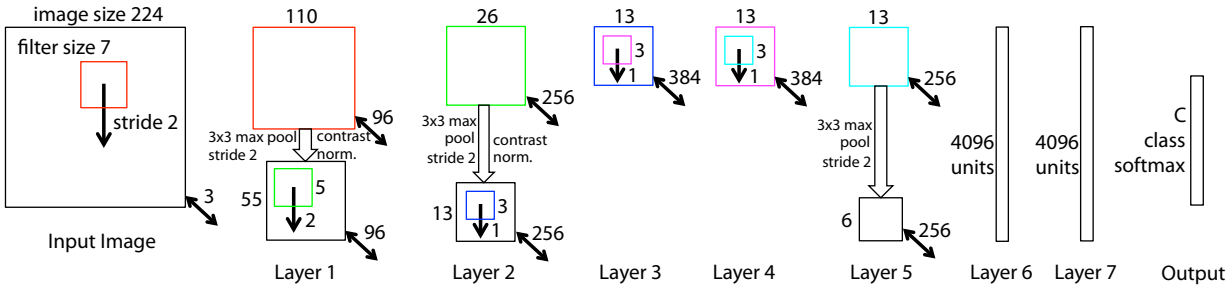
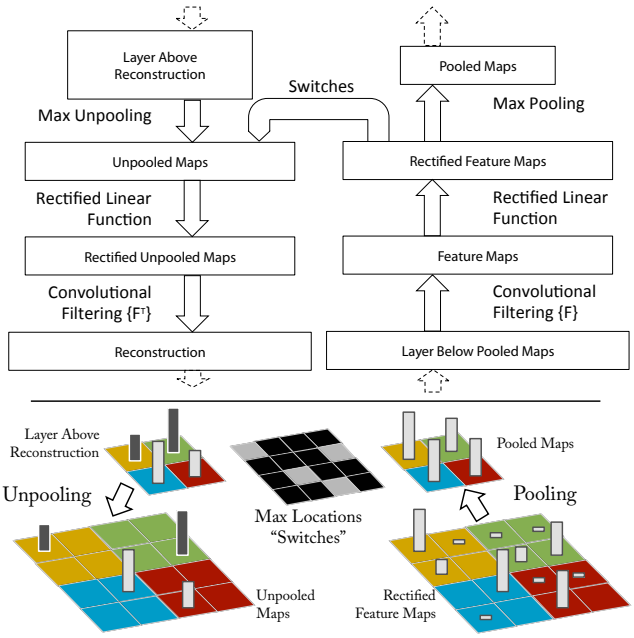



Figure 10: Zeiler's CNN architecture




Every max pooling layer acts as a switch map. The index of the element having max value is stored. During deconvolution, these indexes are used to populate elements of lower layers from values at higher layers. Deconvolution is just the convolution with a transpose of the filter under consideration.

Figure 11: Coupled Convolution and Deconvolution Network

The most interesting point of this paper wasn't the marginal change in architecture, shown in figure 10, but the reason behind it. The authors, in 2010 had published a paper on deconvolutional NN [25], shown in figure 11, which produced the low level stimulus (in layer 0 - image space) that stimulates a particular trained filter at any higher layer. This network was coupled with the AlexNet CNN and preferential stimuli of filters in different layers were analyzed. It was observed that in first layer, most of filters were

sensitive to either high or low spatial frequency, thus mandating a decrease in filter size and stride, so that they can filter out middle frequencies too. This coupling of deconvolutional layer also gave us insight to the kind of filtering that goes on inside a deep-CNN (more details beyond scope of this report).

R-CNN – But all the above networks are sensitive to variation in training and test conditions. The biggest problem is a disconnection between what and where i.e. where in image space is the region that our network has classified as 'x'? To solve this **R-CNN** – Rich Hierarchical CNN [26] from Berkeley, preprocesses a given image and generates nearly 2000 candidate regions which may have an object. This selection can be done by a number of object selection algorithms in CV literature. Each of the regions is fed into the CNN individually and thus resulting in classification of localized regions, as shown in figure 12.

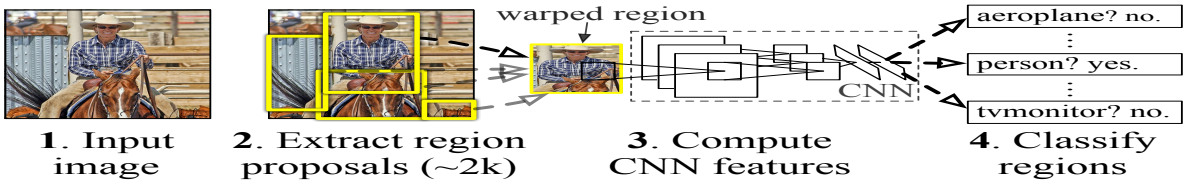



Figure 12: RCNN

Parameter Server – The architecture of CNN is fairly generic. Stacked convolution layers each optionally followed by pooling and reLU layers and finally couple of layers of fully connected RICA- Reconstruction ICA algorithm. When the size of this network is large (line AlexNet with ~60 million parameters), its called as deep neural network. All the above architectures were tightly coupled, i.e these were run in

machines, even though they have multiple cores or GPUs. In [27], [28], Jeffrey Dean, Q.V.Le and Andrew Ng designed a 1000 node machine and ran a CNN on a massive scale and data. The inspiration behind this work was that, accuracy of humans in object recognition isn't just in computation but also the amount of data we experience during learning. Instead the modern deep networks work on a scale of data that is an aorta of what can be naturally experienced. They ran their unsupervised massive network on youtube videos (scaled down to 256x256) for a few days and observed the features learnt by higher layer nodes/neurons using deconvolutional network previously discussed. It was observed that these neurons were sensitive to high-level features such as human faces and figure of cats (which was not surprising considering the plethora of cat videos in youtube). Biologically, these neurons acted the same way as neurons in IT part of the brain (highest functional part in ventral pathway) works, which is sensitive to, among other things, human faces.

Even though the structure of each layer of this network was same as that of a generic CNN, it had about 1 billion parameters. This posed a problem from a computational standpoint. The amount of time required optimizing these many parameters may be months, if run on single machines. Therefore, the authors adopted a distributed computing approach. They used a cluster of 1000 machines each having 16 cores, thus 16000 cores, to train 5 instances of the entire network separately, each instance was again distributed among many machines. The data was distributed and convolution operation tiled such that there was minimal inter-machine communication. Further more, each instance of the network ran a different mini-batch of training data. But as backpropagation is a serial process, weights adjusted by different data would be

different. To avoid this, a parameter server was used which had an exclusive duty of maintaining and updating weights. It was the responsibility of all the instances of networks to send their gradients to the parameter server after forward propagation of each mini-batch, and receive the updated weights before training the next mini-batch. This is called Asynchronous SGD-Stochastic Gradient Descent. The salient features learnt by the high level neurons were observed to be more resilient to rotaion, scale and vertical and horizontal offset than previous networks.

Google Lenet – One of the issues faced by previous designs was the loss of information due to pooling layers which was designed to alleviate multi-scale recognition problems. Also, it is computationally expensive if there are many large filters (like most of the filters were of size 5x5), as these are full matrix operations. These problems were addressed by Google Lenet [29], shown in figure 13, by introducing parallel pipelines with 1x1 convolutions. Their use is two fold- 1. They preserve information which would've rather been lost due to large filter convolutions, 2. They make use of sparsity of layer activation at higher layers. This makes use of [30], which concluded that incase of sparse deep network, we can design better network topology if we have knowledge of the previous layer's probability distribution of activation. But as it is very difficult to analyze/encode the activation statistics of a layer, its best if all the filter sizes are used. Thus there are multiple filters used at any given higher layer, not just for sparsity, but also to preserve details occurring at different scales. Every convolution layer is followed by a reLU operation and preceded by occasional pooling layer. They also used multiple sites of softmax regression/classification layer. This was to avoid the problem of decaying gradients. As height of the network increases its very difficult for

backpropagation error to trickle down to lower layers. Mltiple sites generating classification error (at different layers), added to the final errors propagated from top, make sure this weight decay is not severe. As there are so many parameters in the network, it is susceptible to overfitting, as discussed; therefore the rate of dropout was increased from 50% to 70%.

Finally, there are about 22 layers in this network, but if all the 1x1 layers preceding 3x3 and 5x5 filters are taken into consideration, there are about 100 layers (depending on implementation).

This network performed best in Imagenet 2014 giving a top-5 error of 6.67%. It also used the same technique as RCNN (to be discussed below) for localization alongside classification. But as can be observed the size of this network is a major bottleneck.

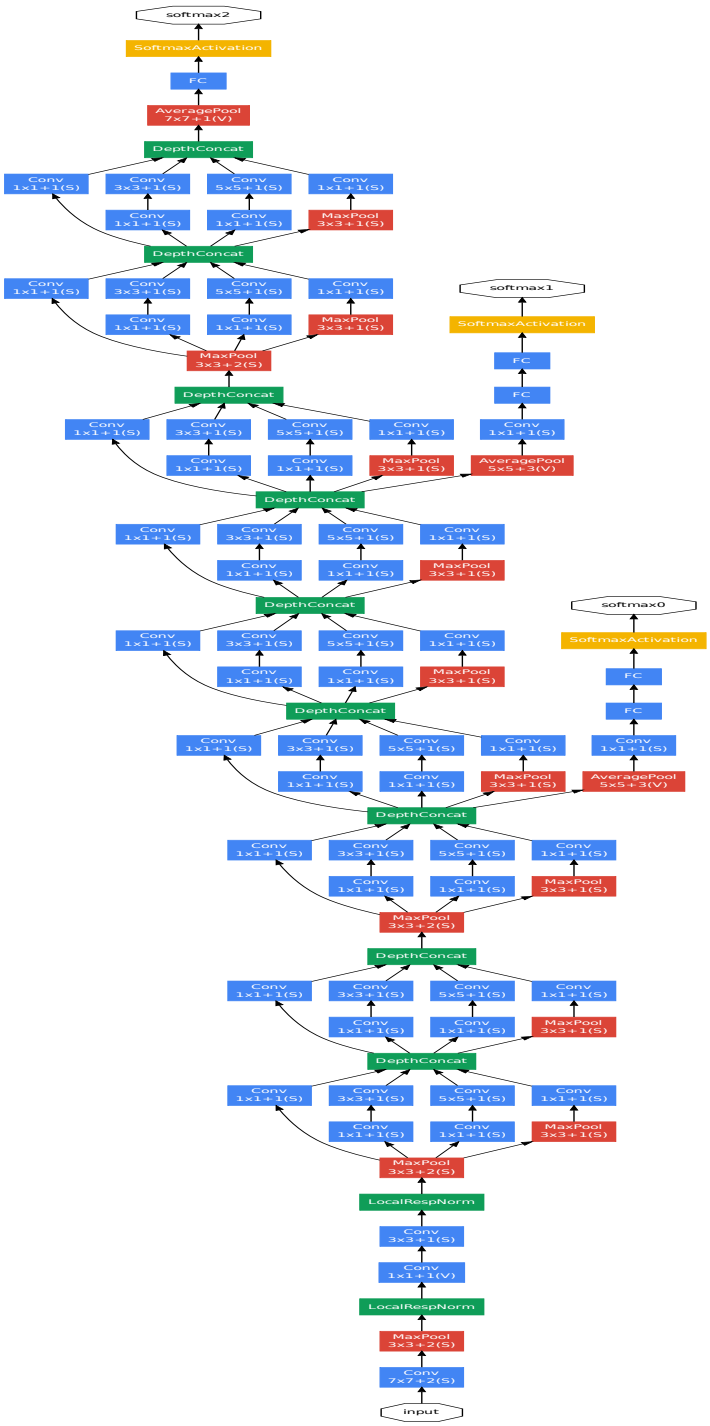



Figure 13: Google LeNet

# CHAPTER 4

# DRAM SIMULATION

As the first part of this thesis, we tried to create a DRAM simulator for studying the behavior of memory access of a CNN algorithm. One of the most popular open source platforms for development of CNNs is Caffe: Convolutional Architecture for Fast Feature Embedding[31] from BVLC, UC Berkeley. We used a C variant of Caffe called 'darknet' from University of Washington. The source code was compiled in CPU-only mode, the network was loaded with AlexNet configuration and the weight of the neurons was equal to learned weights of Alexnet. While all images undergo a pre-processing operation before they can be fed into the CNN, we didn't analyze it as the access pattern is not poignant to the algorithm. This pre-processing includes a image format conversion from jpeg/tiff i.e. padded/unpadded format to standard OpenCV complaint image format such as IPLImage. We know images in jpeg format are stored in RGB format, which is a 3d matrix. Pixels of the three channels are stored discreetly across the third dimension. Incase of padded formats like tiff format, a fourth layer of padded layer is added so that memory accesses are more structured. Whereas opencv format such as IPLImage converts the 3d matrix format of images into a single dimensional array. Pixels of all channels having common 2d co-ordinates are stored in adjacent positions in this format. While this format is beneficial for many image-processing operations, in deep learning systems, pixels in different channels are not processed independently. Furthermore most of the current CNNs have untied weights, and the convolution is carried out in a strided

format i.e. he image is segmented into small windows. These windows have some overlap with one another. Thus if an operation were carried out across the entire image on all windows in parallel, there would be considerable read contention due to this overlap. Therefore, it is better to duplicate overlapping pixels and disassociate windows having overlapping regions. This leads to a nearly 5 times increase in input data in the first layer of Alexnet. As the filters and input image to a convolution layer are both 3 dimensional, all pixels belonging to the same 3d window are stored adjacently in an array.

At the heart of any CNN layer, there is a kernel called gemm- general matrix-matrix multiplication. This kernel is responsible for the convolution operation of a filter with each window of the input image. The inputs to this kernel are instrumented as suggested above. The trace of the DRAM simulator was obtained from this kernel. Unlike popular DRAM simulators such as dsim[32] which need a number of attributes in their memory trace, our trace contained only 3 attributes for each memory access- 1. Operation type- read/write, 2. Array name- A/B/C (convolution layers have 3 inputs- A= weight, B=input image, C= output) and 3. Index of the element being accessed. We also tried using standard memory trace generator such as intel pin tool[33] to generate a trace of the application while running an Alexnet, but as the application was targeted for single core general purpose computers, it instruments and de-instruments the inputs and outputs after every layer's processing, there are many such operations which are not poignant to the algorithm and wouldn't be needed in a dedicated FPGA system, and using a standard memory trace generator would've forced our simulator to analyze those operations too. Therefore, we created our own traces by distilling only those parts of the application,

which are essential to our analysis. A trace was generated for each layer, when the application was running in training mode i.e. for both forward and backward propagation.

## Architecture

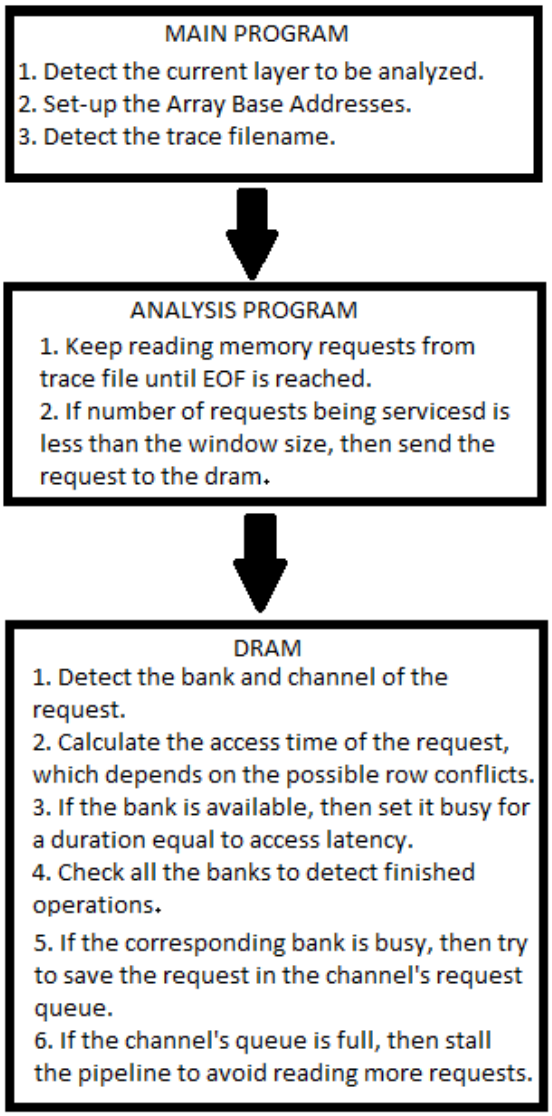



Figure 14: Flowchart of DRAM Simulator

As shown in figure 14, the simulator starts by detecting the current layer to be analyzed and loading the corresponding trace file. This file is then transferred to the

analysis program. This program reads each memory access, which is a triplet as stated above, and keeps sending them to the DRAM simulator program. The analysis program returns control to its caller only when there is no more memory operations in the given file. Before the analysis program can be started, the main program also has to set up the base addresses of each array that will be accessed by the simulator. Currently we are hardcoding the base addresses according to the layer that is being run. This also gives us the freedom to store different arrays in same or different banks/channels. The mapping of array addresses to actual DRAM banks can also be customized before compilation.

The simulator is an approximate simulator that has few essential features of DRAM. This is an asynchronous, trace driven simulator, i.e. there is no concept of a synchronous clock and the virtual clock is incremented when a new memory address is fed into the simulator by the analysis program. There is a caveat, when all the banks as busy, there is a possibility that the simulator may become unresponsive. In such a case, the simulator increases virtual clock while blocking the pipeline to avoid analysis program from reading more memory operations. The simulator also has queue, which are channel specific, and instructions that cannot be processed, because their corresponding bank is busy, are moved into the queue. The analysis program can keep reading new requests as long as the DRAM queues aren't filled. Intuitively before a memory request can be sent into the simulator, all the queues are checked for pending requests and if any of the corresponding request's bank is available, that request is removed from the queue. From the simulator's perspective, processing a memory request is equivalent to just calculating the access time- this depends on which row is open in the corresponding bank and then marking that bank busy for that much time. There is no actual data transfer

anywhere. The timing calculations are similar to SDRAM. The timing parameters in the simulator are not absolute and are relative to the virtual clock, thus we can also analyze the effects of different relative speeds between cpu and memory.

In an attempt to simulate multiple processing sites, there is a concept of window in the analysis program. The total number of requests on the fly in the system cannot be more than the window size. For most of our analyses we kept this number equal to 500. There is also the concept of dispatch rate, which represents the approximate delay between two successive memory requests to the memory. It is the ratio between dispatch rate and timing parameters that simulate the difference in speed between cpu and memory. Here, we didn't take care of any Read after Write, Write after Read or Write after Write dependency as there is no such dependency in CNN algorithm unless both the input and output arrays are mapped to the same memory, which in our case, isn't true. Also there is concept of hierarchical cache in this model. Even if there would be a hit rate of ~95% by adding a single layer of cache, due to prefetching, this model concentrates on DRAM for now.

Also the array elements were stored in a cyclic manner across the banks. As the memory accesses are serial, such a distribution pattern ensured that every new request would go to a different bank than the previous request. Thus ensuring more parallelism while catering to multiple requests on the fly.

## Result

The DRAM simulator, running under two different configurations, analyzed the traces as shown in figure 15 and 16. The two configuration were- single channel single

controller and double channel double controllers. In the later case, we assumed the set-up not only had two channels, but each of these two DRAMs had there own queues. As the trace analysis is very long, we can discuss only very few interesting observations here.

When the ratio between the dispatch rate and timing parameters is high enough, we can start seeing contention in the DRAM. This can be observed by polling the DRAM for the status of its banks and queues once in every ten million operations. Under single DRAM mode, with 64 banks, and a queue size of 64, we can observe that the number of requests on the fly is almost always nearly 120, and there are about 60 requests in the DRAM at any given time. When the memory configuration was changed to have 2 DRAM channels with independent controllers and having the same timing constraints as earlier, it was observed that the number of requests on the fly reduced to about 90. But there was almost no request in any of the DRAM queues under this set-up. When the memory configuration is set-up in 2 channel mode, it is important to note that assigning different arrays to different DRAM controllers is required to ensure parallel servicing of requests. If this is not done, then the result is same as single controller configuration. Also, it was observed empirically that the most of the memory accesses are either belonging to the input image array (B) or the output array (C), the weight array (A) is accessed once every 2916 iterations. Therefore, during set-up of array base address to memory mapping, it was ensured that these arrays are mapped to different channels and controllers. The word size of the memory was equal to 4 bytes as all of the accesses are single precision floating point data. Under the existing timing conditions, there wouldn't have been any new observation if we had increased the number of banks or channels. As the DRAM queue is almost always empty, this means the memory banks are never

saturated in a 2 channel configuration. The only condition when increasing banks and channels would show some improvement in performance is if the ratio between dispatch rate and timing parameters was increased, i.e. memory was made to operate more slower than the processing element(s).

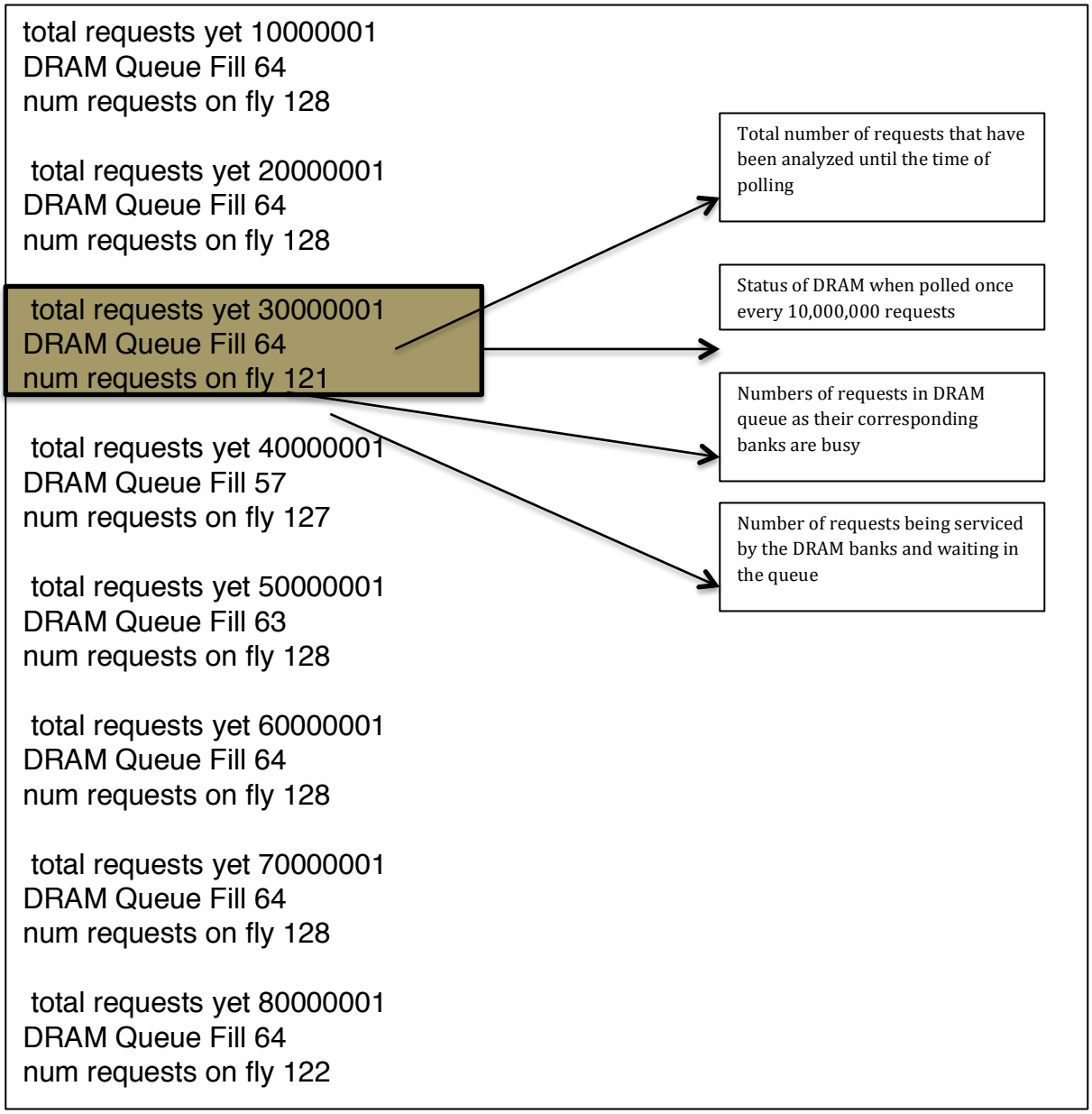



Figure 15: Sample Simulator output of single channel DRAM running on first convolution layer trace

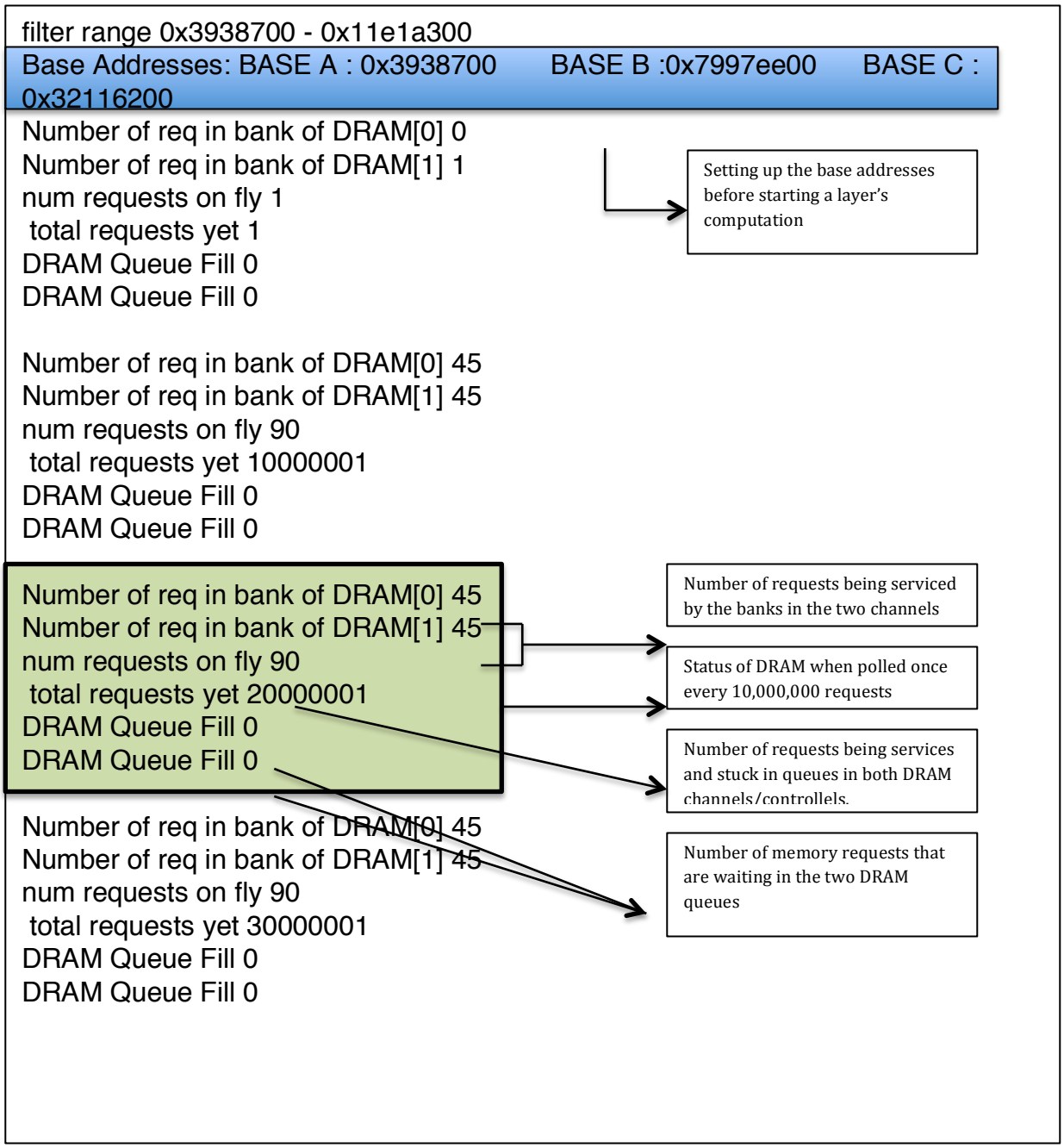



Figure 16: Sample Simulator output of double channel DRAM running on first convolution layer trace

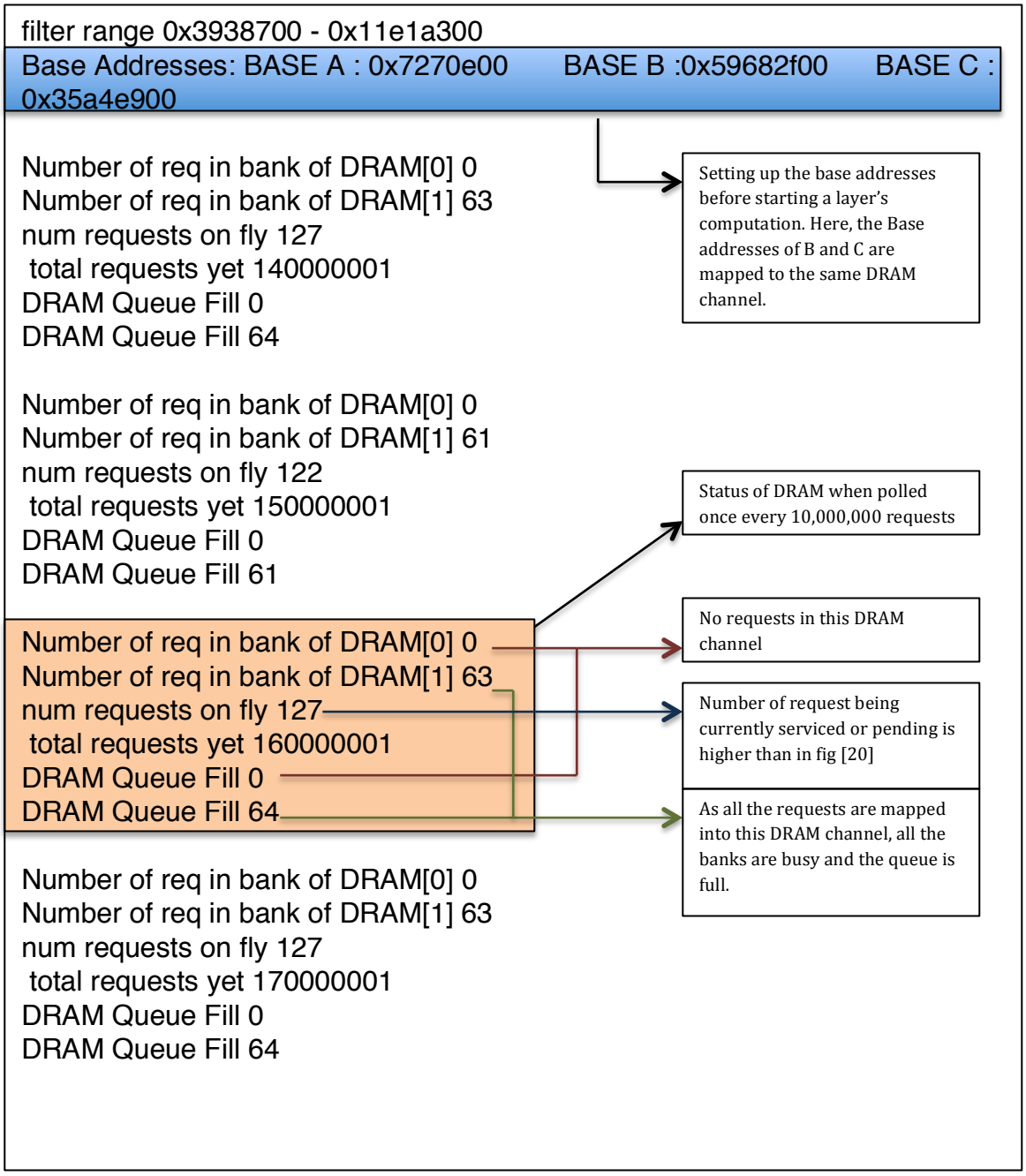



Figure 17: Sample simulator output of second convolution layer trace

It can be observed from the trace simulator that the second convolution layer, shown in figure 17, which takes its input from previous max-pool layer, doesn't reap the benefit from two channels. This is because all the layer outputs are put in single DRAM channel and hence both the input and output are stored in the same DRAM leading to access contention. This effect is also observed in the last three convolution layers of the network, which are adjacent to each other, and their inputs and outputs feed into each other in forward and backward propagation. This contention can be resolved by putting outputs of adjacent layers in different DRAM channels thus creating a ping-pong access where the inputs and outputs of each layer come from different DRAMs thus avoiding any kind of access collision.

It might feel unintuitive to request the DRAM for every access, as mentioned earlier; even the presence of a single layer cache can achieve a hit rate of ~95% due to pre-fetching. But as we are targeting FPGA, the above analysis is helpful if the designed is realized in a very small but fast FPGA that consumes very little power but has a very small or no on-chip memory.

# CHAPTER 5

# FPGA DESIGN OF CNN

In this section we analyze using FPGAs to compute the CNNs. For simplicity purposes we target the design for the first convolution layer only, this is because this layer consumes 19.6% of execution time in a CPU and 16.9% of execution time in GPU[34] during forward propagation and also consumes the highest amount of memory among all layers. Besides the design that works best for this layer will also work for the rest of the convolution layers, as the core operation is the same across the board.

There have been few works on realizing large scale CNN such as Alexnet on an FPGA[35][36], [37]. All these works concentrate on designing FPGAs for forward propagation only, as it's a current practice in industry and academia to train the network once in the beginning and then deploying, which uses only forward propagation. This simplifies the design as the outputs needn't be stored and can be overwritten, thus decreasing the memory footprint. Also, most of these algorithms don't expand the input due to overlapping windows- as prescribed by GEMM. But, we created a design assuming the input image is expanded. Under current norms, the input image expands from 224x224x3 element array to one having 1.05 Million elements as shown in figure 18. All the inputs and outputs to a layer in an FPGA are stored in DRAM, but they are transferred into the FPGA before starting the computation as shown in figure 19. These accesses can be overlapped with the computations as stated in [34].

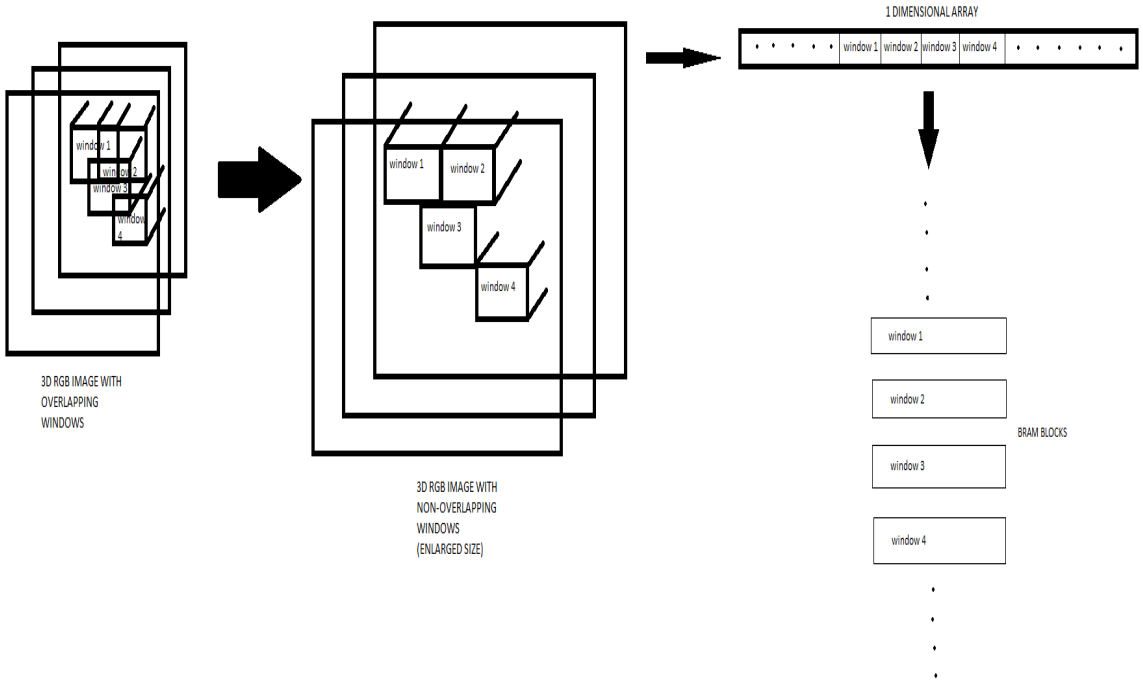



Figure 18: Conversion and storage of input image in BRAM blocks

Our main objective in this design was to increase the number of parallel execution units such that we can, not only fully utilize all the resources in a FPGA, but also find limitations in hardware that can be bottlenecks in increasing parallelism.

## Architecture

The design in figure 20 is similar to [35] as shown in figure 19 but instead of parallelizing across each filter, we parallelize computation across an entire output feature map. We know that convolution is same as reverse correlation, which consists of multiplication and accumulation operations. Each 3d block in the input image is multiplied with each 3d filter and the result is accumulated to produce one pixel of the feature map corresponding to a filter. Under the current configuration, each blob and filter consists of 11x11x3 = 363 elements each. As each of these multiplications don't

have any dependency, they can be carried out in parallel and then the results can be accumulated. Observing the architecture of both [35], [36], it can be inferred they follow similar architecture. This architecture can be further parallelized by operating on a large batch of images than one per iteration. Usually this batch size is kept at 128.

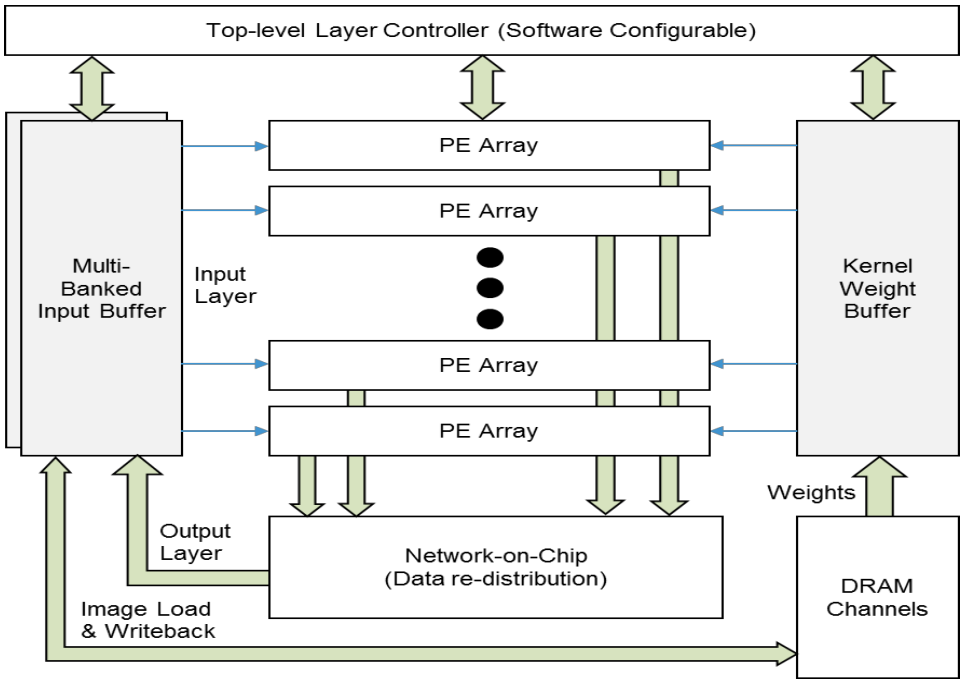



Figure 19: Architecture designed in [33]

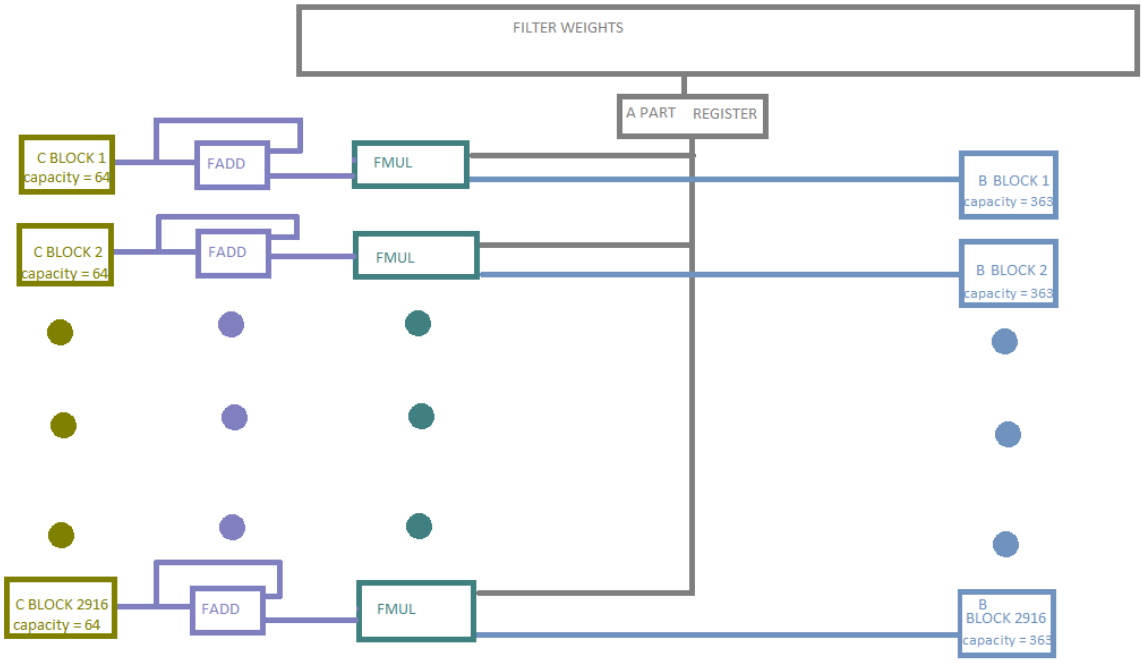



Figure 20: Designed FPGA Architecture

We designed the FPGA for a batch size of 1 image. Once this design is realized we can expand it for bigger batch size, if there are resources left in the FPGA. There are three dimensions along which the design can be parallelized- filter/window size (363), along all the filters (64) or along the output feature map pixels (2916). The last option gives us the highest parallelization. Before starting any operation, the entire expanded image array is copied into the FPGA Block RAM (BRAM). The access latency to bram is equal to 1. Each pixel of each filter is copied once and stored in a register. This value is then multiplied to the corresponding element of each window of the input image in parallel, as the image array is already expanded, there is no contention while scheduling read operation of all these pixels in parallel. As there are 2916 windows and 2916 elements in each feature map, the number of parallel units we have is equal to 2916 operating in parallel per filter per image. In order to support this kind of parallelization,

we have to distribute the inputs and output arrays in a particular manner. The output is stored in BRAM too, and can be transferred out of the FPGA in parallel with computation.

The expanded input image, which is a 1 dimensional array, is stored such that all pixels of a window are stored adjacent to each other and each window is adjacent to its neighbor. Each window/blob is stored in one BRAM block, thus we need at least 2916 blocks to store the entire image. Similarly, each of the outputs of al the parallel execution units have to go to different BRAM blocks, if full parallelizing can be fully extracted. As the output of each unit is a partial output of any feature map, this operation is carried out 363 times per filter to obtain the complete feature map. In each iteration all the elements of the feature map, containing partial accumulations, are read, added with the multiplier result and written back. Even though it might seem the number of memory accesses is very high, the latency is not. In order to ensure low latency, each pixel of a feature map is stored in different BRAM block. But this might lead to inefficient use of memory, as each BRAM block has a capacity of 18K/36K bits and capable of storing much more than one pixel. Thus, the design stored all pixels of all output feature maps having the same 2d co-ordinate/index, in the same block. As we are never commencing two filters at the same time, to avoid read contention, there is never an instance where we access more than element in the same output block in parallel. Thus avoiding write contention and simultaneously saving memory.

As the input and output arrays are stored in on-chip memory, connections between each BRAM block and the parallel executing units can be a wired, thus the internal fan-in and fan-out of the computing unit is very large. As the data-type of the elements is float, they are realized using the DSP blocks in the FPGA. Also, the storage

of filter/weight arrays are neglected in this analysis, as their access occurs once in every 2916 operations, but as we are doing all the operations in a single iteration, these accesses will occur with a duration equal to the lowest iteration latency. This is avoided by storing them in BRAM, but the BRAM capacity is a problem. As these accesses are regular, the memory reads can be pipelined such that barring the first iteration of the lowest loop, we can receive new elements every cycle.

The FPGA design was realized using Vivadio HLS, this was done because of the large scale of the network. Vivadio is responsible for loop scheduling and streamlining the memory accesses to extract the best utilization of the hardware. In [36] the design was optimized using HLS directives such as loop unroll and loop pipeline. But in our design loop unrolling couldn't be realized due to the large number of input/output BRAM banks. Therefore, we had to hardcode the 2916 operations in the lowest loop manually. The same paper also discussed overlapping computation and memory access by dividing the internal memory into buffers, both the input and output memory were divided into 2 parts each. This allowed loading of data in one set of input and output buffers while the other set was used for computation. Unfortunately doing the same in our design isn't simple as have 2916 input and output buffers. Also as each of the BRAM blocks are used during each iteration, its is very difficult to parallelize computation and memory operation. But instead of using single port BRAM, if we can use a dual ported BRAM in Read-Write/Read-Only mode, we can parallelize computation and memory access. Unfortunately this couldn't be realized in current design, as the 'dataflow' HLS directive (used for parallelizing memory access and computation) couldn't succeed, due to large number of banks. Future work can be continued to solve this problem.

## Memory Storage Calculations

In this section, capacity of BRAM blocks is analyzed.

Ideal Scenario

Size of expanded image array B = 2916 x 363 float elements = 1058508 elements

Size of output image (including all feature maps) array C = 64 x 2916 elements = 186624 elements

Total number of elements in Weights/Filters array = 64 x 363 elements = 23232 elements

Total number of elements to be stored = 1268364 elements

Total size of all the elements = 1268364 x 32 bits (assuming single precision float) = 40587648 bits

Hence, the total size required for the first convolution layer is about 39 Mb. This is within the resource budget of Xilinx Virtex-7 Ultrascale FPGA [38]. But practically, under the available BRAM block configurations, the memory requirement for the design is much higher.

Practical Scenario

Number of elements that can be stored in a single 18K BRAM block = 18Kb/32b = 576 elements. According to our memory layout, each BRAM block stores each input image window/blob, which has 363 elements and can easily fit into a BRAM block. Similarly, the design stores all pixels of all output feature maps with same index in the same BRAM block. As there are 64 filters, each output BRAM block has to store 64 elements. Even

though the BRAM blocks are large enough to contain respective workloads, there is a large amount of memory that is wasted. This can be a problem as shown below

Total number of BRAM blocks required for storing B = 2916

Total number of BRAM blocks required for storing C = 2916

Total number of BRAM blocks required = 5832

As total number of BRAM blocks in a Virtex 7 Ultrascale FPGA is 5040, this design can't fit into the FPGA in its current form.

This starvation in memory also implies we cannot compute different filter operations at the same time. As all the filters need to operate on the same image array stored in BRAM blocks, it will cause a read contention at BRAM memory controller. Usually this is avoided by replicating overlapping data, but as the design is already memory starved, it's not practical to replicate the array.

**Utilization**

Our primary design objective was to increase the resource utilization. Comparing the utilization table of [34] table 1, and that of our design in table 2, it can be observed that our design used a lot more resources than [34]. But our design also over-utilizes DSP blocks by about 5 times than what is available. As we declared our BRAM ports exclusively to ensure the requisite memory layout, BRAM usage estimation doesn't appear in the utilization report. Kindly refer to the previous section for the same.

Table 2: Utilization Table in [34]

| Resource | DSP | BRAM | LUT | FF |
|---|---|---|---|---|
| Used | 2240 | 1024 | 186251 | 205704 |
| Available | 2800 | 2060 | 303600 | 607200 |
| Utilization | 80% | 50% | 61.3% | 33.87% |

Table 3: Utilization Table in Virtex 7 Ultrascale xcvu440

| Resource | DSP | BRAM (18K) | LUT | FF |
|---|---|---|---|---|
| Used | 14580 | - | 1046931 | 1239404 |
| Available | 2880 | 5040 | 2518560 | 5037120 |
| Utilization (%) | 506 | - | 41 | 24 |

**Loop Iteration**

According to the synthesis report, the total latency for all the computation of the first layer for each image is equal to 185984 cycles. It doesn't include the time it will take to send the input image into and to take the output array out from the FPGA.

The 3d GEMM algorithm is a 3 fold-nested loop, if there is a single processing element then the loop latency is very large as shown in table 4. But due to manual unrolling and pipelining, it is reduced to 2 fold-nested loops. As shown in table 5, this set-up reduced the execution latency by a factor of 4148 times. Also evident from the

table is the fact that there are 2916 processing elements, all of which are executed once in every 8 cycles. This parallelism is possible only because of the storage pattern of the arrays, this conclusion was derived when using default HLS partial loop unrolling and pipeline directives didn't show any considerable improvement in latency.

Table 4: Loop Latency with single Processing Element

⊟ **Loop**

| | Latency | | | Initiation Interval | | | |
|---|---|---|---|---|---|---|---|
| Loop Name | min | max | Iteration Latency | achieved | target | Trip Count | Pipelined |
| - Loop 1 | 813003968 | 813003968 | 12703187 | - | - | 64 | no |
| + Loop 1.1 | 12703185 | 12703185 | 34995 | - | - | 363 | no |
| ++ Loop 1.1.1 | 34992 | 34992 | 12 | - | - | 2916 | no |

Table 5: Loop Latency with 2916 Processing Element

```
* Loop:
+-------------+--------+--------+----------+-----------+-----------+------+----------+
|             |     Latency     | Iteration|  Initiation Interval  | Trip |          |
|  Loop Name  |   min  |   max  |  Latency |  achieved |   target  | Count| Pipelined|
+-------------+--------+--------+----------+-----------+-----------+------+----------+
|- Loop 1     |  185984|  185984|      2906|          -|          -|    64|    no    |
| + Loop 1.1  |    2904|    2904|         8|          -|          -|   363|    no    |
+-------------+--------+--------+----------+-----------+-----------+------+----------+
```

# CHAPTER 6

# CONCLUSION

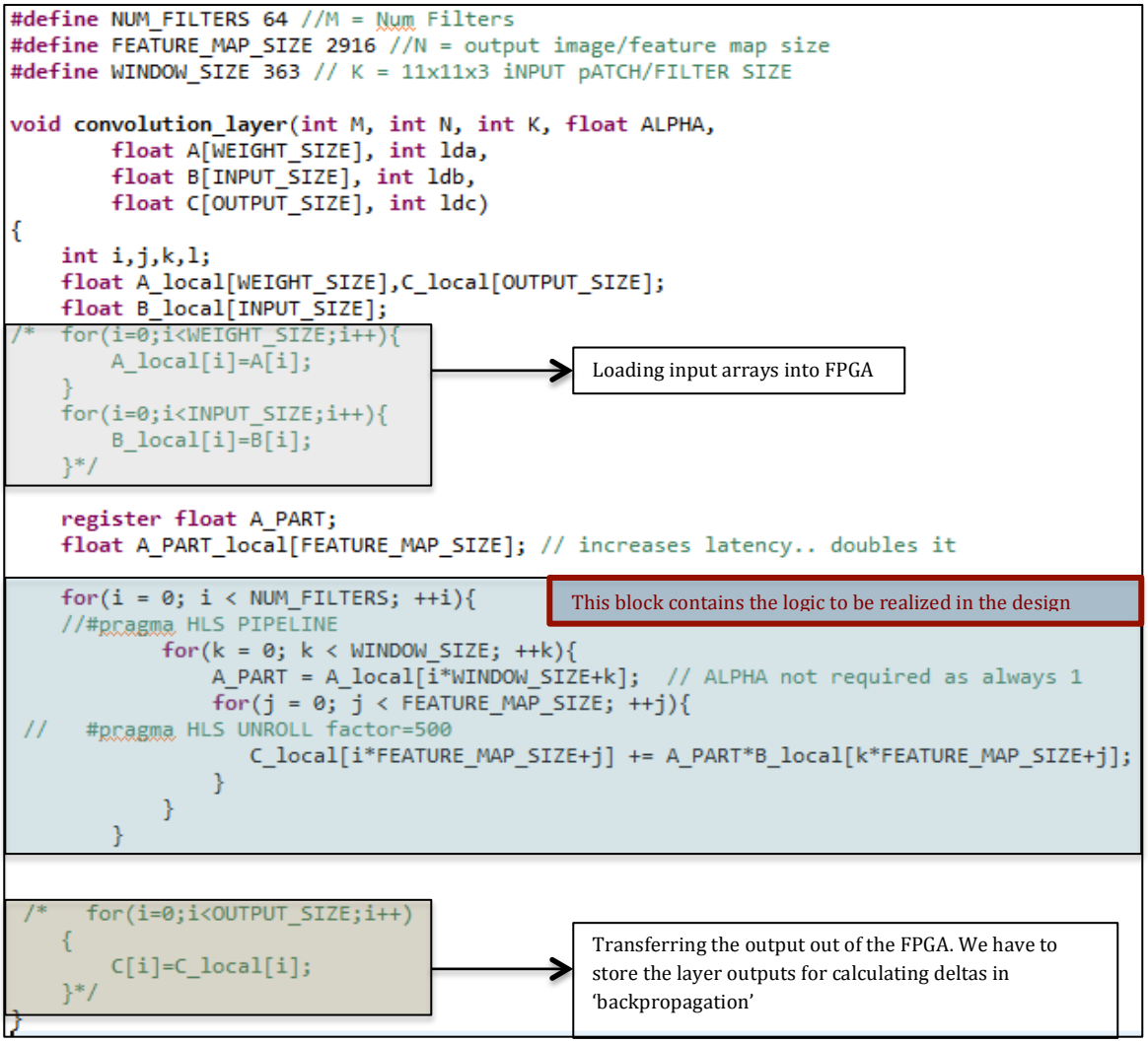
The report explores biologically inspired learning algorithm such as auto-encoders and the systems that are inspired by them. The current neural networks are not remotely close to the scale and complexity of neural circuits found in humans. And the expansion of size is the current trend in AI and deep learning community. Most of the work in both industry and academia use GPUs to run these networks, which are used not only for computer vision tasks such as object recognition but also for natural language processing. A natural recourse for these applications is to be deployed in cloud. Even though current systems are being trained in isolation and deployed in test only mode, i.e. no backpropagation in deployment. But in future, systems will require online learning, which would require backpropagation in production systems. Therefore datacenters should be capable of running both forward and backward propagations in real-time.

While GPUs have performed incredibly by decreasing training and run time of CNNs, their high power usage is a deterrent from being deployed in large-scale systems like in datacenters. The reason behind GPU's success in running neural networks is the presence of many threads execution units that can run in parallel. But as GPUs are designed for graphics operations, each SIMD unit or thread warp unit consists of a bulk of hardware that is not pertinent to run deep learning algorithms. But even if the hardware inside GPU core isn't used they still consume power, but if the hardware can be configured to consist of only pertinent units then we can decrease power usage. This was the motivation behind this thesis. The design that has been realized and synthesized has

increased parallelism beyond the capacity of the state of the art FPGA. From the observations, we can conclude that DSP blocks and BRAM are more important to increase parallelism than flip-flops and Look-up tables, this can guide the design of future generation FPGAs targeted for running deep learning systems.

# APPENDIX A

# CODE WALKTHROUGH




Code Snippet 1: HLS code for FPGA with single Executing Unit

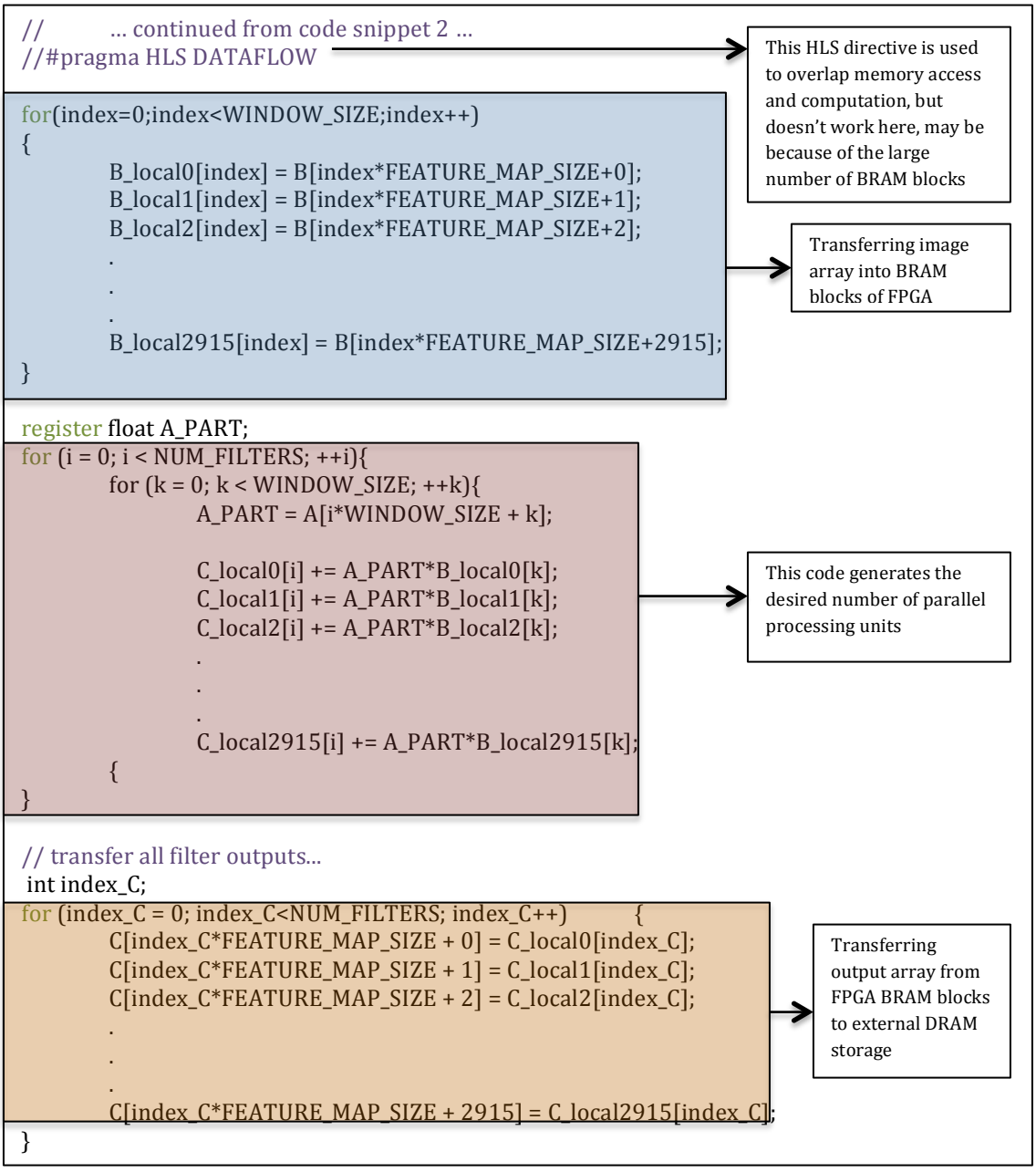
Code snippet 1 shows a sample of the code used for realizing the FPGA accelerator with a single execution unit. Also the snippet contains code to load the inputs into and transferring the output array out of the FPGA. The core logic block of the snippet can be optimized using predefined directives such as 'pipeline' and 'partial

unroll' with a factor of 500, but as explained in FPGA Design – Architecture section, both of these optimization didn't result in desired performance improvement. As there are about 15000 lines of code in the parallel design that was realized, only code snippets of major sections of the program can be shown here.

```
#define NUM_FILTERS 64 //M = Num Filters
#define FEATURE_MAP_SIZE 2916 //N = output image/feature map size
#define WINDOW_SIZE 363          // K = 11x11x3 iNPUT pATCH/FILTER SIZE
void convolution_layer(... ,float A[WEIGHT_SIZE], int lda , float B[INPUT_SIZE], int ldb,
float C[OUTPUT_SIZE],
float C_local0[OUTPUT_SIZE / 2916],
float C_local1[OUTPUT_SIZE / 2916],
.
.
.
float C_local2915[OUTPUT_SIZE / 2916], int ldc,
float B_local0[INPUT_SIZE / 2916],
float B_local1[INPUT_SIZE / 2916],
.
.
float B_local2915[INPUT_SIZE / 2916]) {
        // each C_local<i>[] contains pixel i of all feature maps
        // each B_local<i>[] contains ith blob of input image

        #pragma HLS INTERFACE bram depth=363 port=B_local0
        #pragma HLS INTERFACE bram depth=363 port=B_local1
        #pragma HLS INTERFACE bram depth=363 port=B_local2
        .
        .
        .
        #pragma HLS INTERFACE bram depth=363 port=B_local2913
        #pragma HLS INTERFACE bram depth=363 port=B_local2914
        #pragma HLS INTERFACE bram depth=363 port=B_local2915

        #pragma HLS INTERFACE bram depth=64 port=C_local0
        #pragma HLS INTERFACE bram depth=64 port=C_local1
        #pragma HLS INTERFACE bram depth=64 port=C_local2
        .
        .
        .
        #pragma HLS INTERFACE bram depth=64 port=C_local2915

        // successive code in next snippet...
}
```

Code Snippet 2: Sample code for BRAM port and block declaration

As shown in code snippet 2, to explicitly declare an array to be stored in BRAM block, we need to declare the array both as a function parameter and also as an interface with different ports.




Code Snippet 3: Sample Code containing data transfer and computation logic to generate 2916 parallel processing units

Sample code responsible for data transfer into and out of FPGA and generation of parallel units is shown in code snippet 3. Ideally, we would like to transfer image array into the FPGA and output array out of the FPGA, in parallel with computation of parallel units. This should be possible because we have about 2916 input and output buffers (BRAM blocks) that are dual ported. But the HLS directive responsible to interweave external memory access and internal access and computation doesn't seem to work here, as adding the 'dataflow' directive didn't decrease latency.

# REFERENCES


[1] D. G. Lowe, "Object recognition from local scale-invariant features," *Proc. Seventh IEEE Int. Conf. Comput. Vis.*, vol. 2, 1999.

[2] J. Sivic and A. Zisserman, "Efficient visual search of videos cast as text retrieval," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 4, pp. 591–606, 2009.

[3] S. Fidler and A. Leonardis, "Towards scalable representations of object categories: Learning a hierarchy of parts," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2007.

[4] R. Kasturi, D. Goldgof, S. A. Street, S. College, M. Anderson, M. Peot, M. Aguilar, D. Khosla, Y. Chen, K. Kim, E. Krotkov, D. D. Hackett, G. Technologies, L. Elazary, R. C. Voorhies, and D. F. Parks, "Performance Evaluation of Neuromorphic-Vision Object Recognition Algorithms."

[5] T. Dean, G. Corrado, and J. Shlens, "Three Controversial Hypotheses Concerning Computation in the Primate Cortex," *Twenty-Sixth AAAI Conf. Artif. Intell.*, pp. 1543–1549, 2012.

[6] R. P. Rao and D. H. Ballard, "Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects.," *Nat. Neurosci.*, vol. 2, pp. 79–87, 1999.

[7] D. George, "How the brain might work: A hierarchical and temporal model for learning and recognition," *Learning*, no. June, p. 191, 2008.

[8] J. Hawkins, S. Ahmad, and D. Dubinsky, "HTM Cortical Learning Algorithms,"

pp. 1–68, 2011.

[9] D. George and J. Hawkins, “Hierarchical Bayesian Model of Invariant Recognition in the Visual Cortex Pattern,” *Proceedings. 2005 IEEE Int. Jt. Conf. Neural Networks, 2005.*, vol. 3, pp. 1812–1817, 2005.

[10] Q. Le, A. Karpenko, J. Ngiam, and A. Ng, “ICA with reconstruction cost for efficient overcomplete feature learning,” *Adv. Neural …*, pp. 1–9, 2011.

[11] Y. Bengio, “Learning Deep Architectures for AI,” *Found. Trends® Mach. Learn.*, vol. 2, pp. 1–127, 2009.

[12] L. Deng and D. Yu, “Deep Learning: Methods and Applications,” *Found. Trends Signal Process.*, vol. 7, pp. 197–387, 2013.

[13] G. Hinton, “Deep belief networks,” *Scholarpedia*, pp. 4–5, 2009.

[14] P. Dayan and L. F. Abbott, “Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems,” *Comput. Math. Model. Neural …*, p. 480, 2001.

[15] M. Riesenhuber and T. A. Poggio, “Hierarchical models of object recognition in cortex.,” *Nat. Neurosci.*, vol. 2, no. 11, pp. 1019–1025, 1999.

[16] G. E. Hinton, A. Krizhevsky, and S. D. Wang, “Transforming auto-encoders,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6791 LNCS, no. PART 1, pp. 44–51, 2011.

[17] Q. V Le, J. Ngiam, Z. Chen, D. Chia, P. W. Koh, and A. Y. Ng, “Tiled convolutional neural networks,” *Adv. Neural Inf. Process. Syst. 23*, pp. 1279–1287, 2010.

[18] N. Brady and D. J. Field, “Local contrast in natural images: Normalisation and coding efficiency,” *Perception*, vol. 29, no. 9, pp. 1041–1055, 2000.

[19] V. Nair and G. E. Hinton, “Rectified Linear Units Improve Restricted Boltzmann Machines,” *Proc. 27th Int. Conf. Mach. Learn.*, pp. 807–814, 2010.

[20] X. Glorot, A. Bordes, and Y. Bengio, “Deep Sparse Rectifier Neural Networks,” *Proc. 14th Int. Conf. Artif. Intell. Statisitics 2011*, vol. 15, pp. 315–323, 2011.

[21] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biol. Cybern.*, vol. 36, no. 4, pp. 193–202, 1980.

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2323, 1998.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Adv. Neural Inf. Process. Syst.*, pp. 1–9, 2012.

[24] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,” *arXiv Prepr. arXiv1311.2901*, 2013.

[25] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, “Deconvolutional networks,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 2528–2535.

[26] R. Girshick, J. Donahue, T. Darrell, U. C. Berkeley, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *Cvpr’14*, pp. 2–9, 2014.

[27] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V Le, M. Z. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," *NIPS 2012 Neural Inf. Process. Syst.*, pp. 1–11, 2012.

[28] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng, "Building high-level features using large scale unsupervised learning," 2011.

[29] C. Szegedy, S. Reed, P. Sermanet, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," pp. 1–12, 2014.

[30] S. Arora, A. Bhaskara, R. Ge, and T. Ma, "Provable Bounds for Learning Some Deep Representations," *arXiv Prepr. arXiv1310.6343*, p. 18, 2013.

[31] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, and U. C. B. Eecs, "Caffe : Convolutional Architecture for Fast Feature Embedding," *ACM Conf. Multimed.*, 2014.

[32] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 4, p. 100, 2005.

[33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05*, 2005, p. 190.

[34] Y. Jia, "Learning Semantic Image Representations at a Large Scale," 2014.

[35] K. Ovtcharov, O. Ruwase, J. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware,"

pp. 3–6, 2015.

[36] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks,” *Proc. 2015 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays - FPGA ’15*, pp. 161–170, 2015.

[37] M. Peemen, a a a Setio, B. Mesman, and H. Corporaal, “Memory-Centric Accelerator Design for Convolutional Neural Networks,” *Comput. Des. (ICCD), 2013 IEEE 31st Int. Conf.*, pp. 13–19, 2013.

[38] P. P. Specification and D. Resources, “UltraScale Architecture and Product Overview Summary of Features Processing System,” vol. 890, pp. 1–31, 2015.